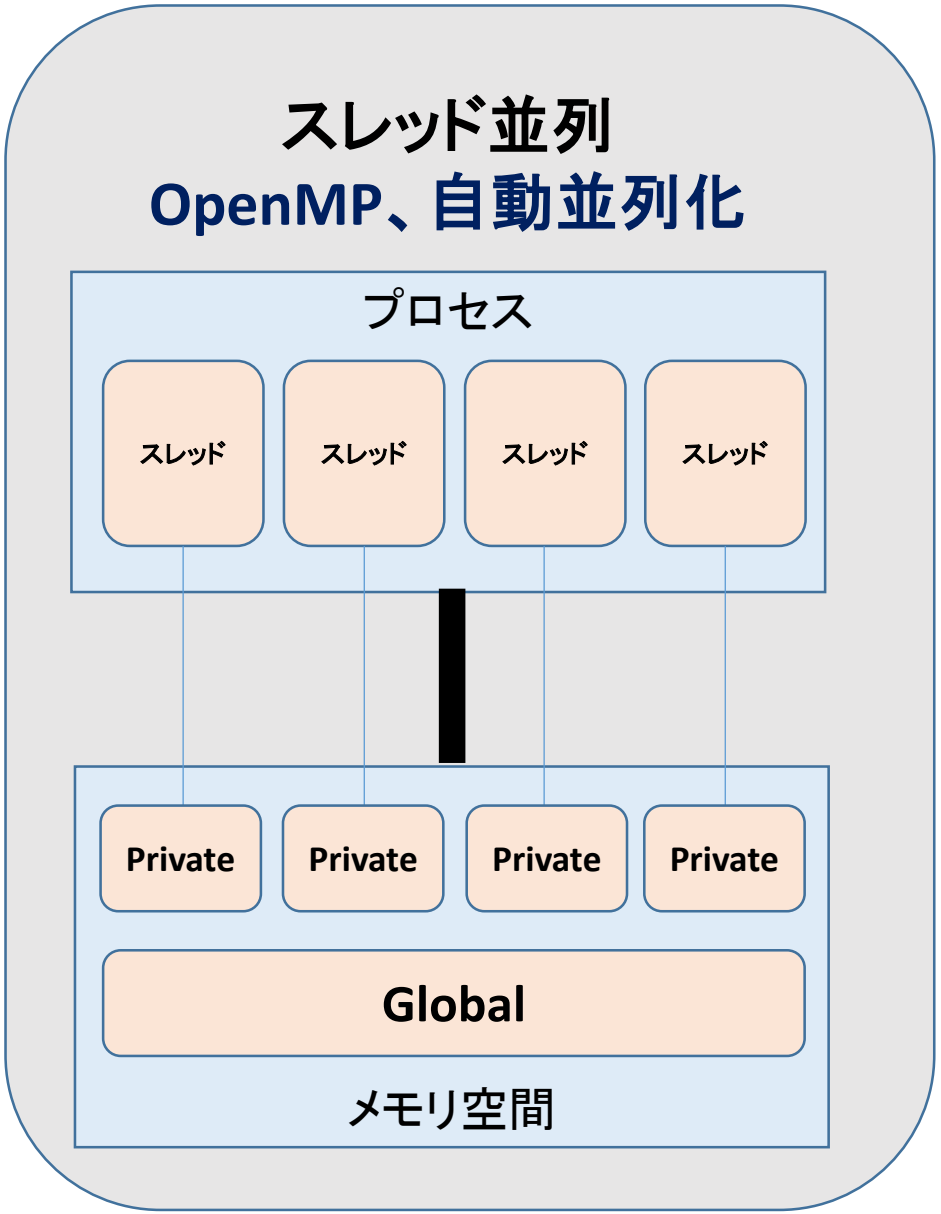
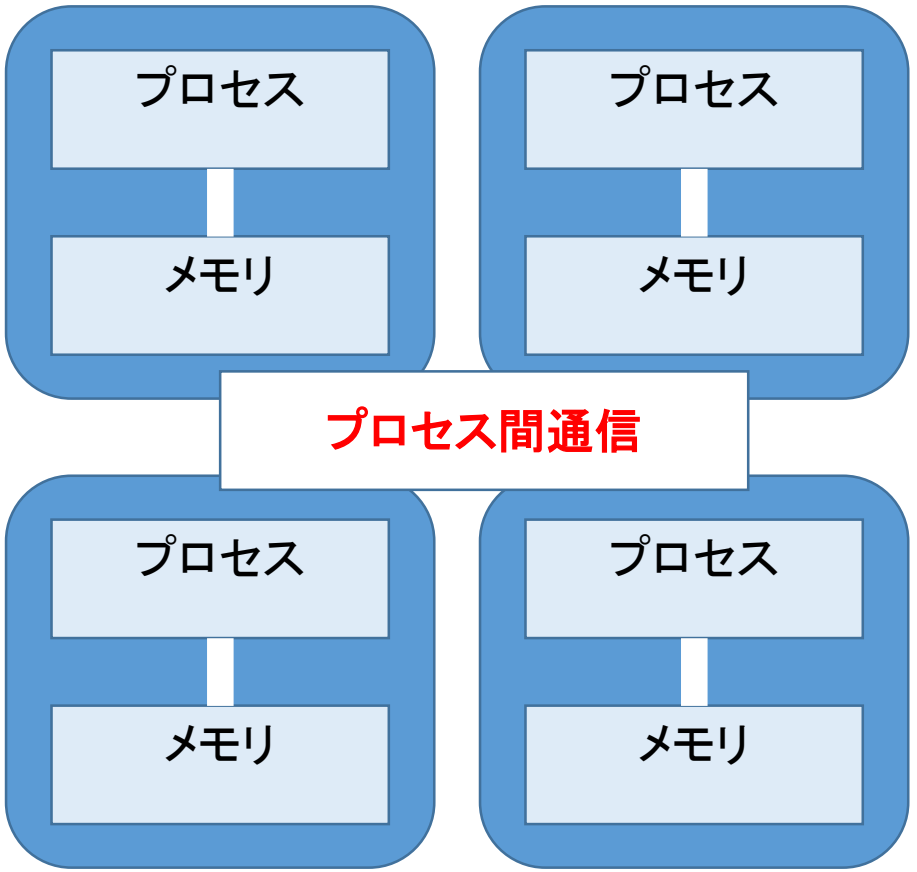
OpenMPによる並列化実装

八木 学
（理化学研究所 計算科学研究センター）

KOBE HPC Spring School 2019
2019年3月14日

スレッド並列とプロセス並列





並列計算について

・CPUをN個使って並列計算した時、計算速度がN倍になるのが理想だが・・・

- 並列化率の問題（アムダールの法則）
- 通信時間ボトルネック

・京のような大型計算機を有効利用するためには、如何に計算速度をN倍に近づけられるかが重要

アムダールの法則

プログラムの並列化できる割合を P とし、プロセッサ数を n とすると、並列計算した時の性能向上率は

$$\frac{1}{(1-P)+\frac{P}{n}}$$

で与えられる。

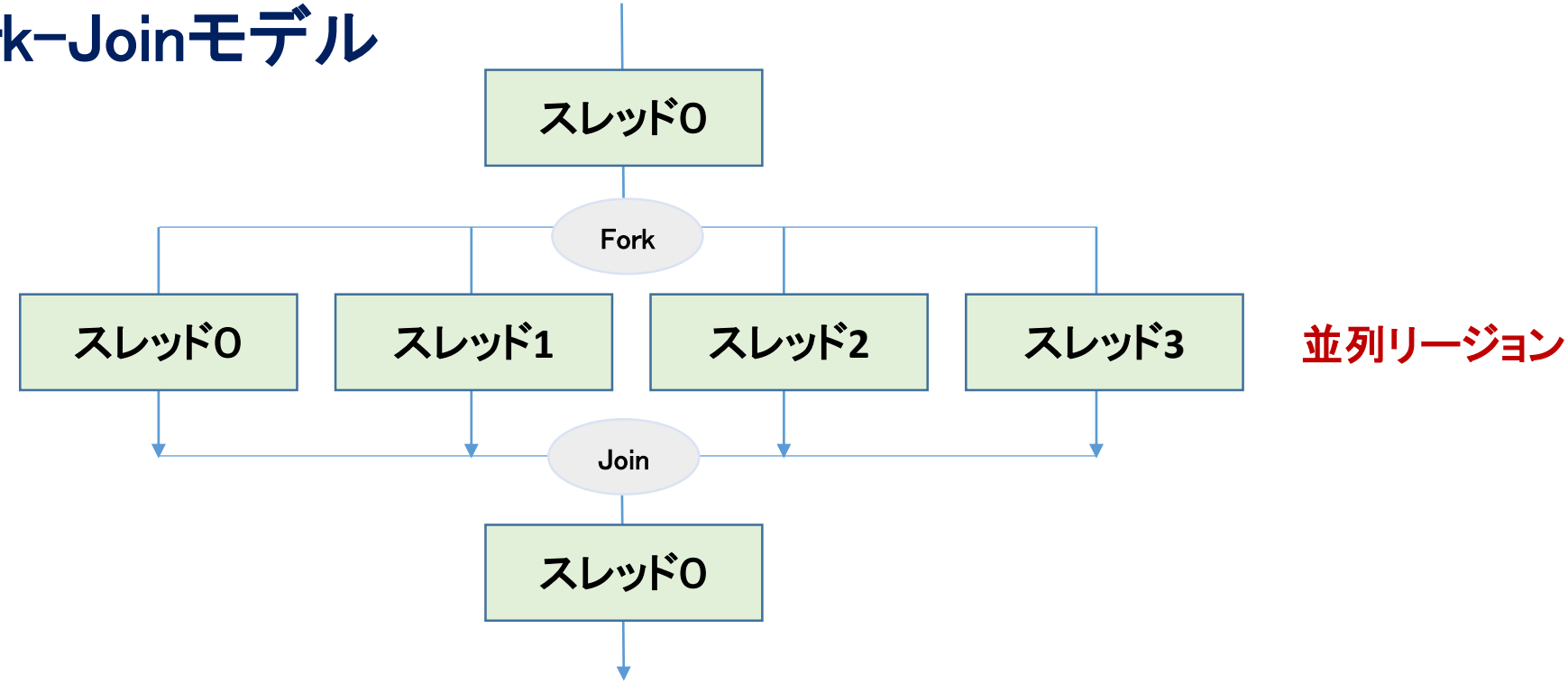
9割並列化できるが、1割逐次処理が残ってしまうような場合、どれだけプロセッサを投入しても計算速度は10倍以上にはならない。

OpenMPとは

- 共有メモリ型計算機用の並列計算API
  →ノード内の並列（ノード間は不可）
- ユーザーが明示的に並列のための指示を与える
  →コンパイラの自動並列とは異なる
- 標準化された規格であり、広く使われている
- 指示行の挿入で並列化できるため、比較的手軽

OpenMPによるスレッド並列

Fork-Joinモデル



F

```
...             非並列
!$omp parallel          Fork
...             並列リージョン
!$omp end parallel      Join
...             非並列
```

C

```
...             非並列
#pragma omp parallel    Fork
{
 ...            並列リージョン
}       Join
...             非並列処理
```

スレッド数の指定

・シェルの環境変数で与える（推奨）

export OMP_NUM_THREADS=16　（bashの場合）

setenv OMP_NUM_THREADS 16　（tcshの場合）

・プログラム内部で設定することも可能

C

```
#include <omp.h>
omp_set_num_threads(16);
```

スレッド数の指定

・シェルの環境変数で与える（推奨）

export OMP_NUM_THREADS=16　（bashの場合）

setenv OMP_NUM_THREADS 16　（tcshの場合）

・プログラム内部で設定することも可能

```
!$use omp_lib
call omp_set_num_threads(16)
```

コンパイル

・コンパイルオプションでOpenMPを有効にする

gcc −fopenmp test.f90

icc −qopenmp test.f90

・オプションを指定しない場合はOpenMPの指示行は無視される。

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    a[i] = b[i] + c;
}
```

C

指示行はFortranの場合 !$OMP から始まる。
行頭の！は通常はコメントを意味する。

Cの場合は #pragma omp という形になるが、
オプションを入れない場合、通常処理されない。

コンパイル

・コンパイルオプションでOpenMPを有効にする

gfortran −fopenmp test.f90

ifort −qopenmp test.f90

・オプションを指定しない場合はOpenMPの指示行は無視される。

```
!$OMP PARALLEL DO
do i = 1, 100
   a(i) = b(i) + c
enddo
!$OMP END PARALLEL DO
```

指示行はFortranの場合 !$OMP から始まる。
行頭の！は通常はコメントを意味する。

Cの場合は #pragma omp という形になるが、
オプションを入れない場合、通常処理されない。

OpenMPの基本関数

OpenMPモジュール/ヘッダをロード

[C] **#include <omp.h>**

[F] **use omp_lib**

*OpenMP関連の関数を使用するためのおまじない

```
!$ use omp_lib
integer :: myid, nthreads

nthreads = omp_get_num_threads()
myid = omp_get_thread_num()
```

F

```
#include <omp.h>
int myid, nthreads;

nthreads = omp_get_num_threads();
myid = omp_get_thread_num();
```

C

OpenMPの基本関数

最大スレッド数取得（Integer）

[C][F] nthreads = **omp_get_num_threads()**

自スレッド番号取得（Integer）

[C][F] myid = **omp_get_thread_num()**

F

```
!$ use omp_lib
integer :: myid, nthreads

nthreads = omp_get_num_threads()
myid = omp_get_thread_num()
```

C

```
#include <omp.h>
int myid, nthreads;

nthreads = omp_get_num_threads();
myid = omp_get_thread_num();
```

OpenMPの基本関数

時間を測る（倍精度型）

[F][C] time = **omp_get_wtime()**

```
!$ use omp_lib
real(8) :: dts, dte
dts = omp_get_wtime()
  ・・・ 処理 ・・・
dte = omp_get_wtime()
print *, dte-dts
```

F

```
#include <omp.h>
double dts;
double dte;
dts = omp_get_wtime();
  ・・・ 処理 ・・・
dte = omp_get_wtime();
```

C

なお、OpenMPモジュール（ヘッダ）のロードを忘れると、これらの関数を使用できずコンパイルエラーになる

Working Sharing構文

○ 複数のスレッドで分担して実行する部分を指定
○ 並列リージョン内で記述する
　#pragma omp parallel { } の括弧範囲内

指示文の基本形式は
[C] **#pragma omp** ***xxx***
[F] **!$omp** ***xxx*** ~ **!$omp end** ***xxx***

◎for構文, do構文
　ループを分割し各スレッドで実行
◎section構文
　各セクションを各スレッドで実行
◎single構文
　1スレッドのみ実行
◎master構文
　マスタースレッドのみ実行

並列リージョンを指定

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<100; i++) {
      a[i] = i
  }

  #pragma omp single
  {
    ...
  }

  #pragma omp for
  for (...)
  .....
  }
}
```

C

複数スレッドで処理
（並列リージョン）

スレッドの起動～終結

[C] #pragma omp parallel { }

括弧 { } 内が複数スレッドで処理される。

for構文

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<100; i++) {
      a[i] = i
  }

 #pragma omp single
  {
    output(a);
  }

  #pragma omp for
  for (i=0; i<100; i++) {
      b[i] = i
  }
}
```

C

forループをスレッドで分割し、並列処理を行う

[C] #pragma omp for

・forループの前に指示行 #pragma omp for を入れる

#pragma omp parallel でスレッドを生成しただけでは、全てのスレッドが全ループを計算してしまう

#pragma omp for を入れることでループ自体が分割され、各スレッドに処理が割り当てられる

1スレッドのみで処理

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<100; i++) {
      a[i] = i;
  }

  #pragma omp single
  {
    output(a);
  }

 #pragma omp for
  for (i=0; i<100; i++) {
      b[i] = i;
  }
}
```

C

[C] #pragma omp single { }

逐次処理やデータの出力のような処理が入る場合、全スレッドで行う必要はなく、1スレッドのみで処理を行えばよい

#pragma omp single { }
を用いることで、{ } 内の記述は1スレッドのみで処理される

並列リージョンを指定

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do

!$omp single
call output(a)
!$omp end single

!$omp do
do i = 1, 100
   b(i) = i
enddo
!$omp end do
!$omp end parallel
```

F

複数スレッドで処理（並列リージョン）

スレッドの起動

[F] !$omp parallel

スレッドの終結

[F] !$omp end parallel

do構文

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do

!$omp single
  call output(a)
!$omp end single

!$omp do
do i = 1, 100
   b(i) = i
enddo
!$omp end do
!$omp end parallel
```

doループをスレッドで分割し、並列処理を行う

[F] !$omp do ～ !$omp end do

・do の直前に指示行 !$omp do を入れる

・enddo の直後に指示行 !$omp end do を入れる

!$omp parallel でスレッドを生成しただけでは、全てのスレッドが全ループを計算してしまう

!$omp do を入れることでループ自体が分割され、各スレッドに処理が割り当てられる

1スレッドのみで処理

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do

!$omp single
  call output(a)
!$omp end single

!$omp do
do i = 1, 100
   b(i) = i
enddo
!$omp end do
!$omp end parallel
```

[F] !$omp single ~
!$omp end single

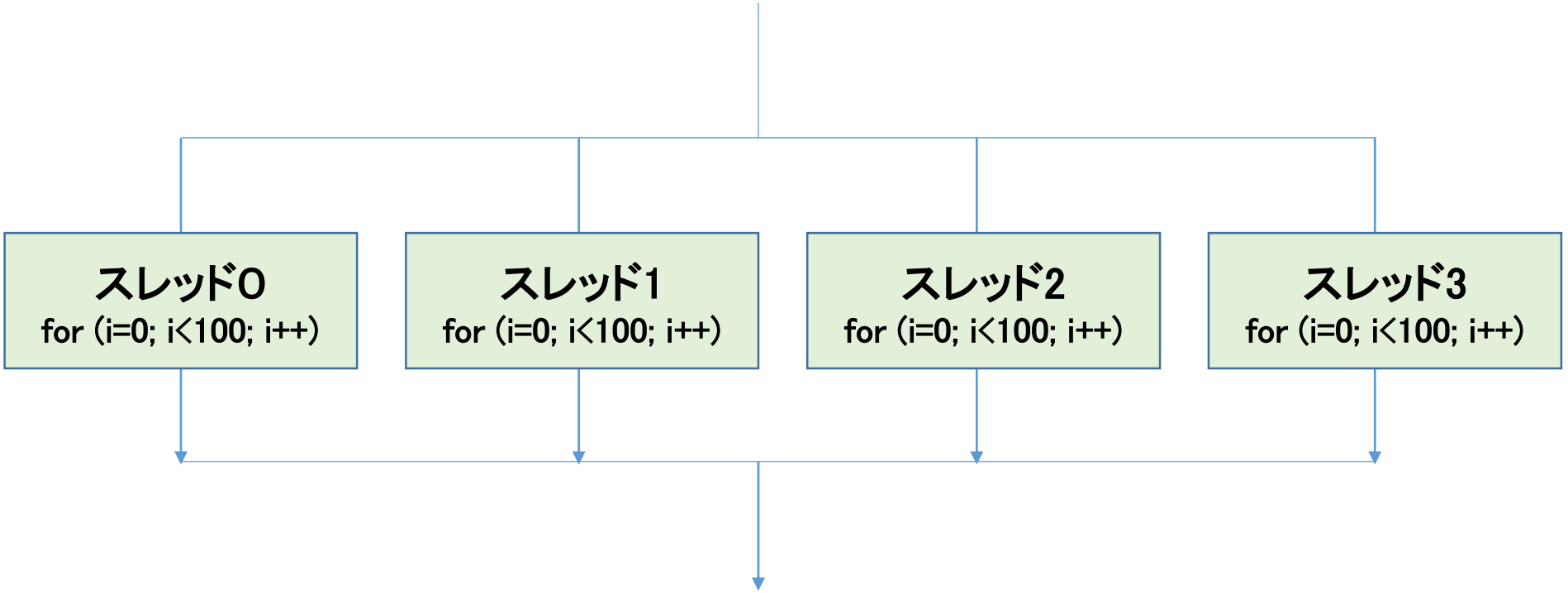
逐次処理やデータの出力のような処理が入る場合、全スレッドで行う必要はなく、1スレッドのみで処理を行えばよい

!$omp single
を用いることで、{ } 内の記述は1スレッドのみで処理される

OpenMPによるスレッド並列

```
#pragma omp parallel
{
  for (i=0; i<100; i++) {
    a[i] = i;
  }
}
```

スレッドを生成しただけでは、全スレッドが全ての処理を行ってしまい負荷分散にならない

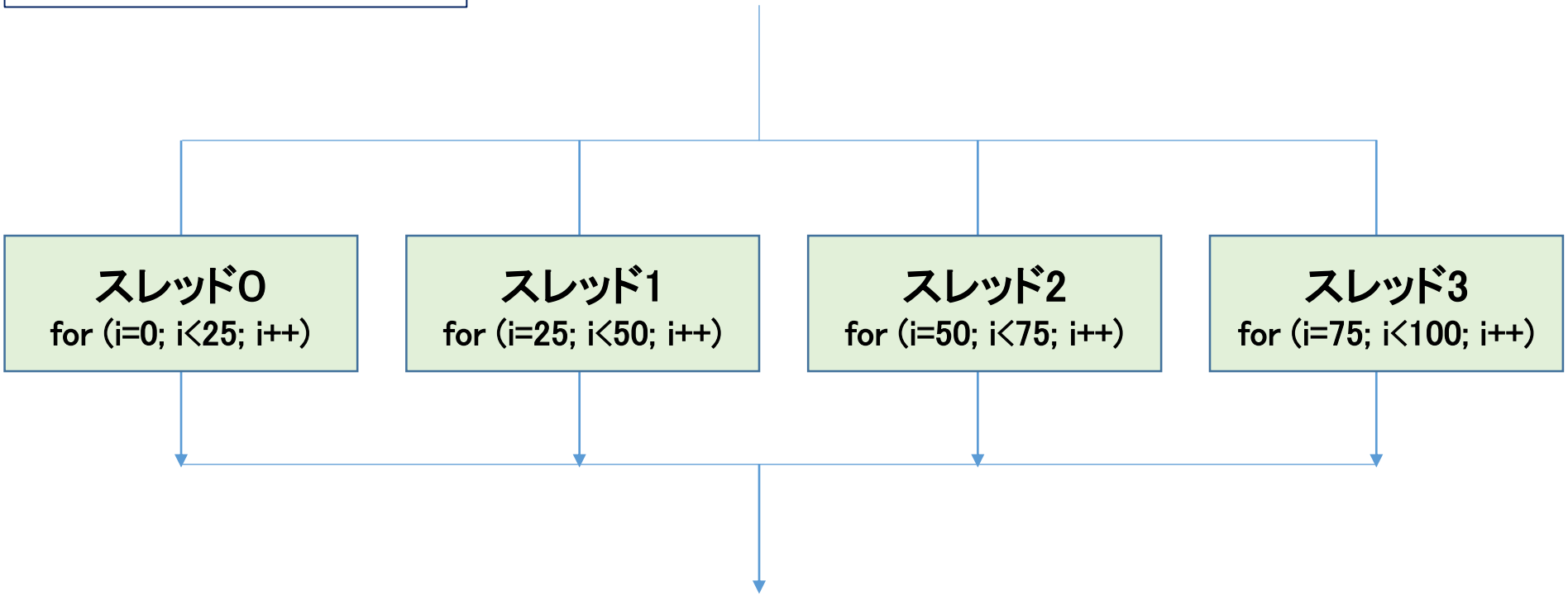


OpenMPによるスレッド並列

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<100; i++) {
    a[i] = i;
  }
}
```

ワークシェアリング構文を入れることにより、処理が分割され、正しく並列処理される。

#pragma omp for、!$omp do はループを自動的にスレッド数で均等に分割する



OpenMPの基本命令

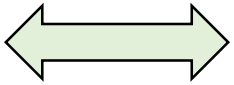
スレッド生成とループ並列を1行で記述

[C言語]

#pragma omp parallel { }
#pragma omp for
→#pragma omp parallel for と書ける

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<100; i++) {
      a[i] = i;
  }
}
```

C



```
#pragma omp parallel for
for (i=0; i<100; i++) {
      a[i] = i;
}
```

C

OpenMPの基本命令

スレッド生成とループ並列を1行で記述

[Fortran]

!$omp parallel

!$omp do

→!$omp parallel do と書ける

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do
!$omp end parallel
```

F

⟺

```
!$omp parallel do
do i = 1, 100
   a(i) = i
enddo
!$omp end parallel do
```

F

プライベート変数について

・OpenMPにおいて変数は基本的には共有（shared）であり、どのスレッドからもアクセス可能である。プライベート変数に指定した変数は各スレッドごとに値を保有し、他のスレッドからアクセスされない。

・並列化したループ演算の内部にある一時変数などは、プライベート変数に指定する必要がある。

・例外的に
[C]#pragma omp for
[F] !$omp parallel do
の直後のループ変数はプライベート変数になる

プライベート変数について

プライベート変数を指定

[C] #pragma omp parallel for private(a, b, ...)

[C] #pragma omp for private(a, b, ...)

```
#pragma omp parallel
{
  #pragma omp for private(j, k)
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
      for (k=0; k<nz; k++) {
        f[i][j][k] = (double)(i * j * k);
      }
    }
  }
}
```

C

ループ変数の扱いに関して

並列化したループ変数は自動的にprivate変数になる。しかし多重ループの場合、内側のループに関しては共有変数のままである。

左の例の場合、i は自動的にprivateになるため必要ないが、j, k についてはprivate宣言が必要となる。

プライベート変数について

プライベート変数を指定

[F] !$omp parallel private(a, b, ...)

[F] !$omp do private(a, b, ...)

```
!$omp parallel
!$omp do private(j, k)
do i = 1, nx
  do j = 1, ny
    do k = 1, nz
      f(k, j, i) = dble(i * j * k)
    enddo
  enddo
enddo
!$omp end do
!$omp end parallel
```

F

ループ変数の扱いに関して

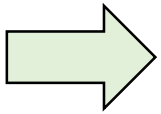
並列化したループ変数は自動的にprivate変数になる。しかし多重ループの場合、内側のループに関しては共有変数のままである。

左の例の場合、i は自動的にprivateになるため必要ないが、j, k についてはprivate宣言が必要となる。

プライベート変数について

起こりがちなミス

```
#pragma omp for
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```



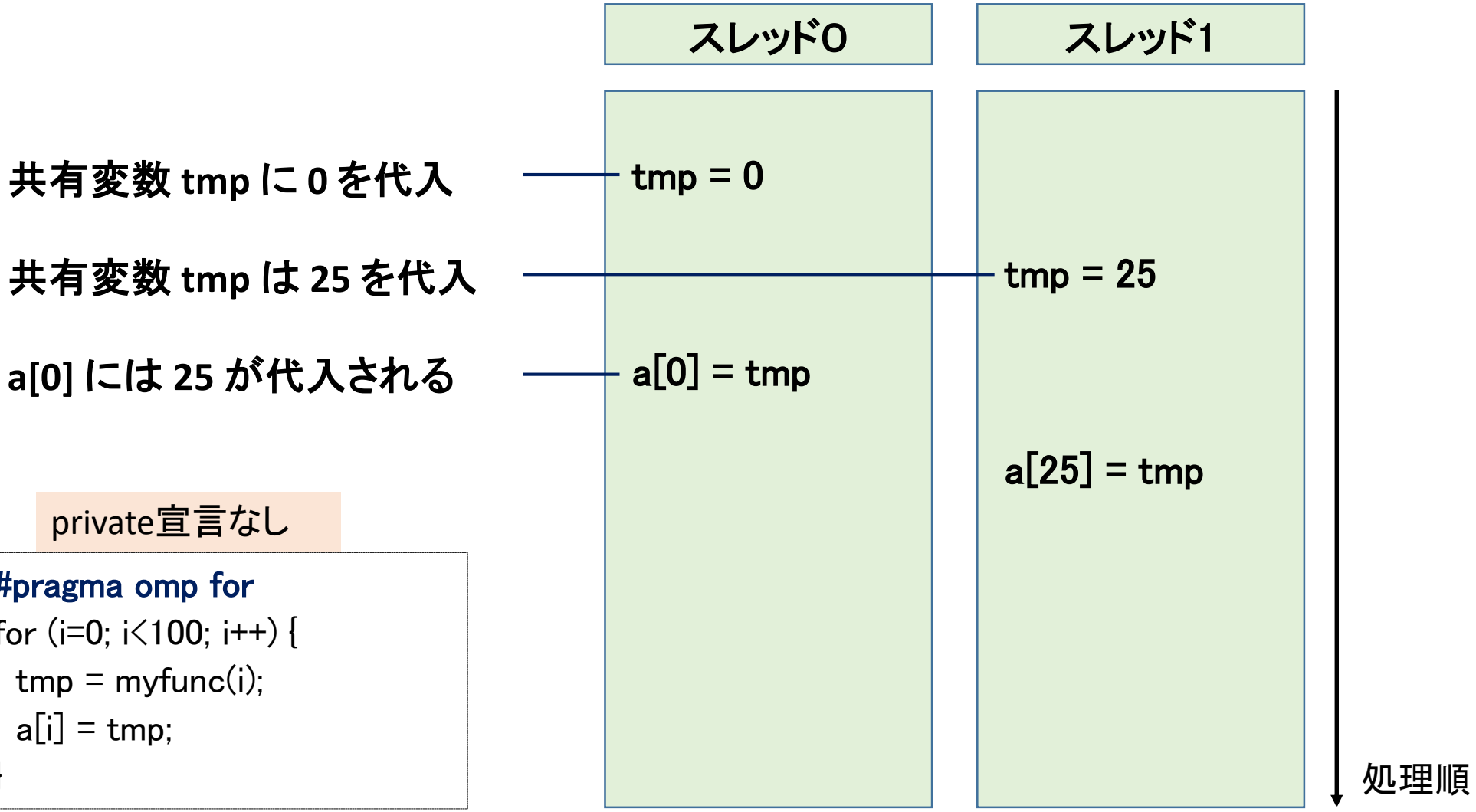
```
#pragma omp for private(tmp)
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```

tmpを上書きしてしまい、
正しい結果にならない

private宣言を入れる

並列化したループ内で値を設定・更新する場合は要注意
→privateにすべきではないか確認する必要あり

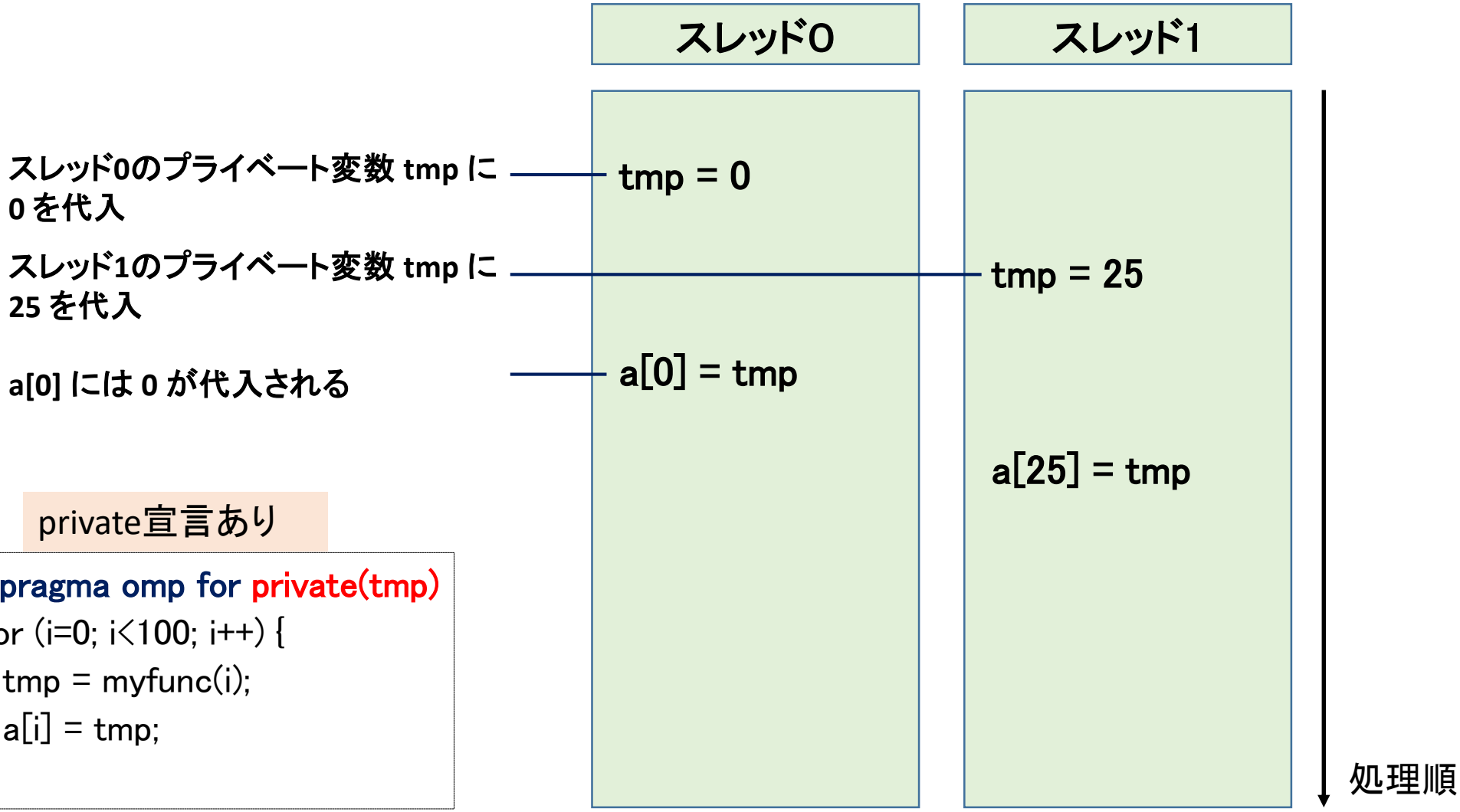
プライベート変数について



private宣言なし

```
#pragma omp for
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```

プライベート変数について



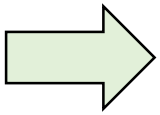
private宣言あり

```
#pragma omp for private(tmp)
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```

多重ループに関して

良くない例

```
for (i=0; i<nx; i++) {
  for (j=0; j<ny; j++) {
    #pragma omp parallel for private(i, j, k)
    for (k=0; k<nz; k++) {
      f[i][j][k] = (double)(i * j * k);
    }
  }
}
```



改善案

```
#pragma omp parallel
{
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
      #pragma omp for private(i, j, k)
      for (k=0; k<nz; k++) {
        f[i][j][k] = (double)(i * j * k);
      }
    }
  }
}
```

OpenMPを用いた並列化では、内側ループ、外側ループのどちらを並列化しても動作はするが、内側ループを並列化すると毎回スレッドの生成を行うため遅くなる。（上記の例では nx * ny 回のスレッド生成）

なお、並列化するループを変えたり、ループの計算順序を変更する可能性があるため、private宣言にはループ変数も書いた方が無難。

共有変数について

共有（shared）変数を指定

[C] #pragma omp parallel shared(a, b, ...)
[C] #pragma omp for shared(a, b, ...)

[F] !$omp parallel shared(a, b, ...)
[F] !$omp do shared(a, b, ...)

・指定しなければ基本的に共有変数であるため、省略可能。

スレッドの同期

nowait を明示しない限り、ワークシェアリング構文の終わりに自動的に同期処理が発生

スレッドの同期待ちをしない

[C] #pragma omp for nowait

[F] !$omp do ~ !$omp end do nowait

スレッドの同期をとる

[C] #pragma omp barrier

[F] !$omp barrier

注意点等

・自動並列化と違い、並列化できるかどうかの判断はプログラマが行う

・依存関係などにより並列化できないループであっても、明示してしまえば並列化されてしまう。

・スレッド内でのグローバル変数、プライベート変数を間違えるとRunごとに結果が変わってしまう。

→数回実行し、結果が変わらないことを確認

・OpenMPは手軽だが、デバッグには注意が必要

演習問題2-1

OpenMPを用いて次のコードを完成させよ

Fortran： 2019spring/code/f90/openmp/1d_adv_omp.f90

C言語： 2019spring/code/c/openmp/1d_adv_omp.c

*演習1にて使用したコードを用いても良い

移流が出来たら流体にも挑戦！

1d_fluid_rk　　：　1次元流体

2d_fluid　　：　2次元流体

補足：マシン上の操作

コンパイル・ジョブの投入（C言語）

> icc **-qopenmp** 1d_adv_omp.c

> qsub run.sh

run.sh

```
#!/bin/bash
#PBS -q S
#PBS -l select=1:ncpus=8          使用ノード数、CPU数
#PBS -N OpenMP                    ジョブ名
#PBS -o output                    標準出力の出力先ファイル
#PBS -j oe
source /etc/profile.d/modules.sh
module load intel                 Intelコンパイラ環境のロード
export OMP_NUM_THREADS=8          スレッド数を指定
export KMP_AFFINITY=disabled

cd ${PBS_O_WORKDIR}
dplace -x2 ./a.out                実行
```

補足：マシン上の操作

コンパイル・ジョブの投入（Fortran）

> ifort **-qopenmp** 1d_adv_omp.f90

> qsub run.sh

run.sh

```
#!/bin/bash
#PBS -q S
#PBS -l select=1:ncpus=8          使用ノード数、CPU数
#PBS -N OpenMP                    ジョブ名
#PBS -o output                    標準出力の出力先ファイル
#PBS -j oe
source /etc/profile.d/modules.sh
module load intel                 Intelコンパイラ環境のロード
export OMP_NUM_THREADS=8          スレッド数を指定
export KMP_AFFINITY=disabled

cd ${PBS_O_WORKDIR}
dplace -x2 ./a.out                実行
```

演習問題2-1

解答例

Fortran：

2019spring/code/f90/openmp/sample/1d_adv_omp_sample.f90

2019spring/code/f90/openmp/sample/1d_fluid_rk_omp_sample.f90

2019spring/code/f90/openmp/sample/2d_fluid_omp_sample.f90

C言語：

2019spring/code/c/openmp/sample/1d_adv_omp_sample.c

2019spring/code/c/openmp/sample/1d_fluid_rk_omp_sample.c

2019spring/code/c/openmp/sample/2d_fluid_omp_sample.c

演習問題2-2

OpenMPを用いて並列化したプログラムを、スレッド数を変えて実行し処理時間を計測せよ

スレッド数の指定方法：スクリプト run.sh の
export OMP_NUM_THREADS=xx　の数値を変更

処理時間の計測

```
dts = omp_get_wtime()
  ・・・　処理　・・・
dte = omp_get_wtime()
print *, dte-dts
```